Solving the advection PDE on the Cell Broadband Engine

Georgios Rokos, Gerassimos Peteinatos, Georgia Kouveli, Georgios Goumas, Kornilios Kourtis and Nectarios Koziris
Computing Systems Laboratory
National Technical University of Athens
Email: {grokos,gpeteinatos,gkouv,goumas,kkourt,nkoziris}@cslab.ece.ntua.gr

***Abstract*—In this paper we present the venture of porting two different algorithms for solving the two-dimensional advection PDE on the CBE platform, an in-place and an out-of-place one, and compare their computational performance, completion time and code productivity. Study of the advection equation reveals data dependencies which lead to limited performance and inefficient scaling to parallel architectures. We explore programming techniques and optimizations which maximize performance for these solver versions. The out-of-place version is straightforward to implement and achieves greater raw performance than the in-place one, but requires more computational steps to converge. In both cases, achieving high computational performance relies heavily on manual source code optimization, due to compiler incapability to do data vectorization and efficient instruction scheduling. The latter proves to be a key factor in pursuit of high GFLOPS measurements.**

***Keywords*-explicit memory hierarchy; Cell Broadband Engine; advection equation; vectorization; parallelization; instruction scheduling;**

I. Introduction

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in diverse areas and fields. These applications are often implemented using iterative finite difference techniques which sweep over a spatial grid, performing nearest neighbor computations called stencils. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space, thereby representing the coefficients of the PDE for that data element. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement (AMR) methods [1].

Stencil codes, such as Jacobi and Gauss–Seidel, are among the most time consuming routines in many scientific and engineering applications. The performance of these codes critically depends on the efficient usage of computational resources, memory bandwidth and local store capacity and can be improved by techniques like double buffering, SIMD vectorization, instruction scheduling and tiling [2]. Usually, all hardware and code comparisons focus on a performance basis. An important aspect of these algorithms, however, is not only the performance expressed in FLOPS that can be achieved by modern CPUs but also the number of computations needed in order to reach a specific accuracy. This last parameter is an important factor at determining the overall time needed for an algorithm to complete execution, which, when dealing with real life computations, is a more important factor than raw FLOPS performance.

In this paper we evaluate these two metrics (raw FLOPS performance and overall completion time) on the Cell Broadband Engine for two versions of the advection PDE solver. The first (out-of-place) version is inherently parallel, straightforward to implement but numerically requires more computational steps to converge, since it follows a stencil scheme that uses values from the previous time steps, although newly computed values are available. The second (in-place) version utilizes newly computed values and is thus more serial requiring significant re-engineering and programming effort to expose parallelism. However, it requires half the time steps of the out-of-place version to converge. The question is which of these two approaches is more appropriate for the Cell B/E, taking into consideration the overall completion time of each version together with the programming effort required for the implementation. In order to have a more solid view on the performance of the final code, we apply code optimization in an attempt to attain the best performance possible. The optimizations investigated include multi-buffering techniques, re-arrangement of elements in memory to reduce transfer overheads, data vectorization, manual instruction scheduling and can be applied to a wide range of compute-intensive applications written for this specific platform and yield respective performance benefits.

The rest of the paper is organized as follows: Section II provides a basic background the reader should be familiar with, describing the algorithms and the specific hardware platform which we used in our experiments. Section III gives technical details and performance models concerning the implementation of these algorithms along with a list of code optimizations and other design choices we have made. In Section IV we present the results of our experiments, focusing on performance measurements and algorithm comparisons. We conclude this paper with general conclusions and ideas for future work in Section V.

II. Background

This study is dominated by the concepts of the advection PDE, algorithms for solving it, the tiling technique, which is used in parallel algorithms, and our target platform: the Cell Broadband Engine.

A. Algorithmic background

Advection equation: In constructing a grid model for many physics problems, one generally faces the problem of solving the advection equation in two or more dimensions. The advection equation is a partial differential equation (PDE) that governs the motion of a conserved scalar as it is advected by a known velocity field and it is widely used in modeling of contamination problems in water, air and soil environments. It is derived using the scalar's conservation law, together with Gauss's theorem, and taking the infinitesimal limit. The equation is

$$\frac{\partial v}{\partial t} = \vec{a}\nabla v$$

where $\vec{a}$ is the velocity field vector. In two dimensions, the above equation is analyzed as

$$\frac{\partial v}{\partial t} = a_x\frac{\partial v}{\partial x} + a_y\frac{\partial v}{\partial y}$$

where the velocity vector $\vec{a}$ has components a_x and a_y in x and y directions respectively.

In order to study the advection in a $X \times Y$ field for a T time window we choose to divide the field in a uniform and specific grid. For this division we use Δt time step and Δx and Δy space steps respectively. We can use Euler–forward method where time derivative is replaced by

$$\frac{\partial v}{\partial t} = \frac{v_{ij}^{n+1} - v_{ij}^{n}}{\Delta t}$$

and space derivation by

$$\frac{\partial v}{\partial x} = \frac{v_{ij}^{n} - v_{(i-1)j}^{n}}{\Delta x}.$$

If we use a united computing grid in both dimensions ($\Delta x = \Delta y$ and $a_x = a_y = a$) the two-dimensional PDE is expressed as

$$v_{ij}^{n+1} = \left(1 + 2a\frac{\Delta t}{\Delta x}\right)v_{ij}^{n} - a\frac{\Delta t}{\Delta x}\left(v_{(i-1)j}^{n} + v_{i(j-1)}^{n}\right) \quad (1)$$

This way the problem forms a numerical analysis problem.

In order to get a result which converges to the actual solution, multiple iterations on the grid are needed. The number of iterations depends on the desired accuracy level, grid size and executed algorithm. In our study, two algorithms have been tested and compared: an in-place algorithm and an out-of-place algorithm.

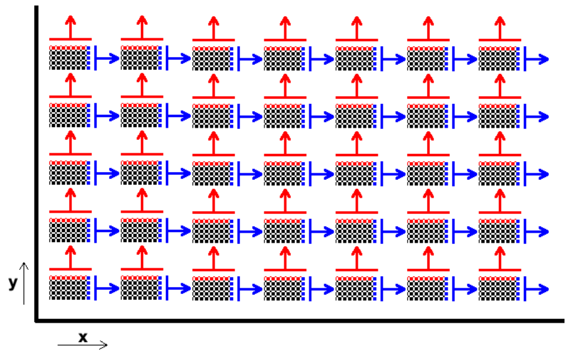

Figure 1. Exchange of border values between processing units

Tiling transformation: Splitting computations along many processing units requires intermediate exchange of border values between these units. Execution of an algorithm across multiple time steps allows for more optimization opportunities due to increased data reuse [3]. In this context, neighboring iteration points are grouped together to build a larger computation node that can be atomically executed without any intervention. Data exchanges are also grouped and performed with a single message for each neighboring processor, at the end of each atomic supernode execution (see Figure 1). Supernode partitioning (or tiling) of the iteration space was proposed by Irigoin and Triolet in [4]. The use of a communication function that has to be minimized by linear programming approaches was proposed by Boulet et al. in [5]. They calculated the total communication produced by a tile as function of its sides and shape and proved that minimization can be done independently of tile volume.

B. The Cell Broadband Engine

The target processor is the Cell Broadband Engine (CBE), which is highly recommended for scientific applications [6], [7]. The Sony–Toshiba–IBM (STI) Cell processor is a heterogeneous, nine-core architecture that combines considerable floating point resources with a power-efficient, software-controlled memory hierarchy. It is the first implementation of the Cell Broadband Engine Architecture (CBEA). It provides increased potential for high performance application development and execution and stands out from conventional processors due to some special architectural characteristics of its coprocessing cores and its memory interface mechanism.

Instead of using identical cooperating commodity processors, CBE uses a conventional high performance PowerPC core that controls eight simple SIMD (single instruction, multiple data) cores called Synergistic Processing Elements (SPE). These computation-oriented cores operate on 128-bit vectors, so an instruction can process either four single-precision words or two double-precision words simultaneously.

Another key feature of each SPE is the three-level, software-controlled memory hierarchy. Instead of transferring data between the 128 registers and DRAM via a cache hierarchy, load and store operations may access only a small (256KB in the first CBEA implementation) private local store (LS). The CBE utilizes explicit DMA operations to move data from main memory to the local store of each SPE. Dedicated Memory Flow Controllers (MFC) allow multiple concurrent DMA loads/stores to be in progress simultaneously with the SIMD execution unit, thereby mitigating memory latency overhead via double-buffered DMA data transfers. All parts communicate with each other through a high-bandwidth (204.8 GB/s) Element Interconnect Bus (EIB) [8], [9]. This approach allows potentially more efficient usage of the available memory bandwidth, but increases the complexity of the programming model.

The Cell processor is designed to deliver an extremely high single-precision performance of 25.6 GFLOPS per SPE (204.8 GFLOPS collectively); however, double precision performance lags significantly behind with only 1.83 GFLOPS per SPE (14.63 GFLOPS collectively), for a 3.2 GHz core clock. The XDR memory interface supplies 25 GB/s peak aggregate memory bandwidth [10]; thus, the most important limiting factor is double precision arithmetic rather than DRAM bandwidth. Trying to explore the full potential of the CBE, the single-precision approach was used in our experiments.

III. Implementation

Implementing a solver for the advection PDE involves choosing appropriate algorithms, making platform-dependent optimizations in order to gain benefits in execution speed and analyze performance models of these algorithms in the context of the underlying hardware in an attempt to understand which techniques can contribute in achieving high performance.

A. Out-of-place advection solver

The out-of-place advection solver is a straightforward implementation of Equation (1), where calculation at each point only depends on values calculated during the previous time step. In our case, the calculation of v_{ij}^{n+1} requires the values of the previous time step v_{ij}^{n}, $v_{(i-1)j}^{n}$ and $v_{i(j-1)}^{n}$. This approach exposes a significant amount of parallelism of since the computations within an entire time step can be legally performed in parallel. The pseudocode of the out-of-place advection solver is given in Algorithm 1.

B. In-place advection solver

The in-place advection solver is numerically faster since the calculation at each point uses two values that have already been computed in the current time step and one value computed in the previous time step. Thus, the calculation of v_{ij}^{n+1} depends on the values of v_{ij}^{n}, $v_{(i-1)j}^{n+1}$ and $v_{i(j-1)}^{n+1}$.

Algorithm 1 Out-of-place advection solver

```
while(!converged())
{
  n = (++loops)%2;

  for(i = 1; i < Y; i++)
    for(j = 1; j < X; j++)
      U[1-n][i][j] = (1+2*a*dt/dx)*U[n][i][j] -
        a*dt/dx*(U[n][i-1][j]+U[n][i][j-1]);
}
```

The in-place algorithm converges with double the rate of the out-of-place algorithm [11]. Its pseudocode is given in Algorithm 2.

Algorithm 2 In-place advection solver

```
while(!converged())
{
  n = (++loops)%2;

  for(i = 1; i < Y; i++)
    for(j = 1; j < X; j++)
      U[1-n][i][j] = (1+2*a*dt/dx)*U[n][i][j] -
        a*dt/dx*(U[1-n][i-1][j]+U[1-n][i][j-1]);
}
```

For both algorithms, the number of operations needed in order to complete a time step is the same, although the total number of time steps significantly varies. Numerically, the version to choose is the in-place one, due to better convergence rate. However, we also need to evaluate two additional important factors: the execution time of each time step and the implementation effort of each method for our execution platform.

C. Optimizations

Stencil computations impose restrictions and limitations during their execution. Many approaches have been developed to deal with this problem and yield a good performance for stencil computations [2], [12]–[14]. To improve computational performance of the selected algorithms, we examine a wide variety of optimizations including multi-buffering, SIMD vectorization, manual instruction scheduling and tiling.

Multi-buffering: Cell uses a cacheless local-store architecture. Moreover, instead of prefetching or multithreading, DMA is the architectural paradigm utilized to express memory level parallelism and hide memory latency. This has a secondary advantage in that it also eliminates superfluous memory traffic from cache line fill on a write miss. Unlike cache-based architectures, we can implement the dual-circular queue approach on each SPE. Input data are split into two buffers. Each time we make a computation block based on data from one buffer, a DMA transfer is executed, prefetching data for the next computation block to the second buffer. We double-buffer both read and write operations.

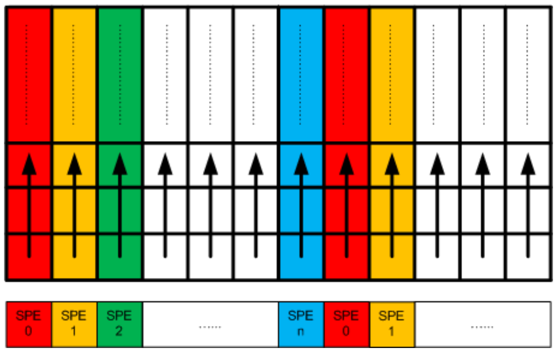

Figure 2. Assignment of blocks to SPEs

Blocks: In the current implementation of CBE, each SPE has a 256 KB local store (LS). In order to process the grid, we have to divide it into smaller blocks that fit the LS. Making the rather arbitrary assumption that this LS should be equally distributed between code and data, all buffers for storing blocks in the LS should not exceed 128 KB. If the double or quadruple buffering technique is implemented, we have to allocate two, respectively four, buffers for storing LS blocks. Additionally, single-precision floating-point vectors consist of 4 elements; therefore the block dimension should be a multiple of 4. Furthermore, we need some space to store the 'borders' that are transferred between SPEs. This space is multiplied by the tiling factor, in case we use tiling.

Taking these parameters into account and after experimenting with multiple tiling factor – block size combinations, we empirically selected an efficient block size of 16 KB, which translates to 4 K single-precision elements. This is also equal to the maximum transfer size for a single DMA transfer allowed by the MFC. The most obvious form of the LS block would be a square, 64×64 block. This choice applies for the out-of-place algorithm. Due to limitations explained below, this approach cannot be applied to the in-place algorithm.

After dividing the grid into blocks, each SPE will be assigned whole block-columns for computation, as shown in Figure 2. This way, vertical border values remain inside the LS of each SPE instead of being transferred between SPEs, reducing bandwidth usage. We know that interprocessor communication is a serious factor in performance degradation in parallel systems, so reducing dependence vectors is essential.

In the out-of-place algorithm a further optimization has been used. Instead of the typical C-style row-major layout of arrays, we can use a block-major layout for the grid. This means that all elements of the same block are stored in consecutive memory addresses. This way, when a block is transferred to and from the LS, this transfer is executed within a single DMA transfer, so communication overhead is minimized and EIB bandwidth is better utilized. Otherwise, in the conventional row-major layout, each row of the block has to be transferred using its own, dedicated DMA transfer and this may lead to EIB congestion (e.g. a 64×64 block requires 64 distinct DMA transfers).

Vectorization: The PowerPC Processor Element (PPE) supports both scalar and vector (SIMD) data types, while the SPEs are exclusively SIMD processors. Although scalar code is supported by the compiler, its performance is significantly degraded compared to the theoretical peak performance. Thus, in order to exploit the full potential of the SPEs, it is critical to vectorize our code. SIMDized code is expected to double performance in double-precision and quadruple performance in single-precision arithmetic, since it executes the same operation in a vector of 2, respectively 4, operands simultaneously, instead of wasting these resources. Data vectorization in the out-of-place algorithm is easily accomplished, since all data needed for computation of the k^{th} time step have been calculated during the $(k-1)^{th}$ time step.

Vectorization in the in-place algorithm cannot be implemented in a straightforward way. Calculating a point U[i][j] in the k^{th} time step depends on data that are calculated in the same time step, U[i-1][j] and U[i][j-1]. These data are adjacent to U[i][j], resulting to a cross linking of input data and results in the same vector. In order to avoid this cross linking, a diagonal traversal of blocks can be arranged. This solution fixes the dependency problem between data but ruins the ability to load vectors from the local store in a direct way, i.e. data forming a vector are not in consecutive memory addresses, but rather scattered.

In an effort to accumulate adjacently computed data to neighboring memory positions, we perform a block transformation from C-default row-major layout into a diagonal-major layout before the first time step (and, of course, a reverse transformation after executing all time steps). This transformation is applied to each block separately, i.e. the grid follows the block-major layout and each block follows the diagonal-major layout; this layout is depicted in Figure 3. This way, each block is traversed in a diagonal way, making it feasible to neighbor data which are adjacently computed. Apart from that, the block-major layout of the grid implies that every block can be transferred between main memory and local storage within a single DMA operation.

Moreover, diagonal traversal of blocks implies that the first and last few diagonals in every block cannot be exactly split into full vectors. As it can be seen in Figure 4, every diagonal in the lead-in area has got one more element compared to previous one, whereas in the lead-out area every diagonal has got one element less than previous one. Having diagonals, the size of which is not multiple of vector size, causes vectorization problems and varying length of diagonals forces us to write special code (instead of a simple loop) to handle lead-in and lead-out areas. In order to suppress the extent of these problematic areas, a more elongated block shape must be adopted, instead of the typical

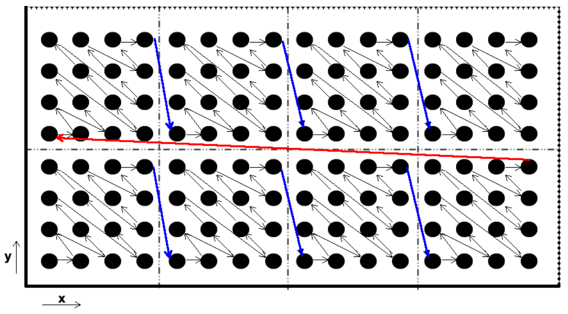

Figure 3. Layout of elements in main memory for the in-place algorithm

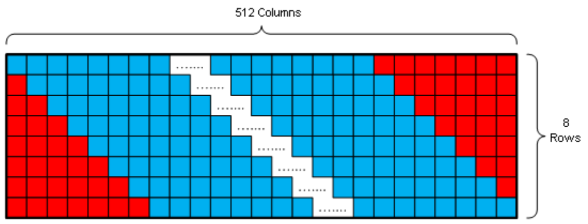

Figure 4. Data block in the in-place method

square form. Keeping the 4 K elements choice in mind, candidates for block dimension are 4×1024 and 8×512. The first choice may seem better, but the second one allows for better exploitation of SPE pipelines. A block dimension of 8×512 has been used in our code.

Instruction scheduling: The CBE has the ability to execute different type of commands simultaneously in two heterogeneous pipelines. The odd pipeline executes data-transferring instructions (load, stores, shuffles etc.), whereas the even pipeline handles arithmetic and logical operations. The SPU processes doubleword-aligned instruction pairs called fetch groups one at a time, in program order. Each instruction in the group can be issued when all its dependencies are met and no structural hazards exist. Dual-issue occurs when both the instructions of a fetch group are issueable, given that the first can be executed on the even pipeline and the second on the odd pipeline. If the instructions cannot be dual-issued, they are issued one after another when they become issuable. Afterwards, a new group is fetched.

In order to maximize performance by exploiting dual-issue, the two pipelines have to be carefully utilized: the odd pipeline has to load and format data in time (i.e. the instruction has to be issued at least 6 clock cycles earlier) so that the even pipeline is continuously fed with required data, without stalling, therefore reaching peak floating-point performance. This careful utilization requires extensive loop-unrolling, not only in order to eliminate expensive branches but also to increase data availability. Unfortunately, this task is not efficiently accomplished by the compiler used in our experiments (GCC v4.3.4) and has to be done manually by the programmer, a task which can prove to be quite painful and time consuming.

Additionally, data dependencies in the in-place algorithm limit the ability to fully exploit the heterogeneous pipelines. Each vector (4 floats) in a diagonal requires 3 consecutive SIMD floating-point instructions, each one having a latency of 6 cycles. The second instruction depends on results from the first one and the third instruction depends on results from the second one. Therefore, the overall latency for computation of each vector is 18 cycles. Having a 8 x 512 block size, diagonals consist of 8 elements, i.e. they form two vectors that can be simultaneously computed. Having this pair of vectors partially exploits the even pipeline, producing 2 vector-results per 18 cycles. For performance enhancement, an explicit instruction scheduling and arrangement has to be made, resulting in the reduction of 18 cycles down to 12 for each vector pair. Yet, this is far from being considered optimal, as the ideal case would be producing 2 vector-results every 6 clock cycles only (2 vectors $\times$ 3 SIMD instructions per vector $\times$ the ability of the even pipeline to produce 1 vector-result per clock cycle).

D. Performance models

Theoretical machine peak performance for single-precision arithmetic is 25.6 GFLOPS per SPE for a 3.2 GHz core clock. This is the result of 3.2 GHz $\times$ 1 instruction completion / clock cycle $\times$ 2 floating-point operations per element $\times$ 4 elements / vector. CBE can perform two floating-point operations per vector element within a single instruction (e.g. multiply two elements and add the result to a third element).

If we use the simple expression $U[n+1][i][j] = C_0 * U[n][i][j] - C_1 * U[n+1][i-1][j] - C_2 * U[n+1][i][j-1]$, which is a more generic solution to the PDE, problem-specific theoretical machine peak is 5/6 of general theoretical machine peak. In both in-place and out-of-place algorithms, computation of a single element consists of one multiply (mul) and two multiply-add (madd) instructions ($temp_0 = mul\,(C_0, U[n][i][j])\,, temp_1 = madd\,(C_1, U[n+1][i-1][j], temp_0)\,, U[n+1][i][j] = madd\,(C_2, U[n+1][i][j-1], temp_1)$). This way, only 5 floating-point operations (out of a maximum of 6, if we could use three madd instructions) are executed per 3 instructions, resulting in the aforementioned drop in problem-specific theoretical peak performance.

- Data dependencies in the in-place algorithm further hinder high performance, causing the 5 aforementioned operations per vector to be accomplished per 6 rather than 3 clock cycles. Therefore, theoretical algorithm peak for this implementation cannot be expected to exceed 5/12 ($\approx$41.7%) of theoretical machine peak, i.e. $\approx$10.7 GFLOPS per SPE.
- There are no data limitations in the out-of-place algorithm, since all data needed for computing the cur-

rent time step have been exported during the previous time step. Therefore, we can get these 5 operations completed per 3 cycles, leading to the conclusion that theoretical algorithm peak coincides with problem-specific theoretical machine peak, i.e. ≈21.3 GFLOPS per SPE.

From the above analysis, using the in-place algorithm seems to yield insufficient performance at first sight. Yet, it is known that the in-place algorithm for solving systems of linear equations is faster, i.e. requires fewer steps to converge, and more reliable than the out-of-place algorithm. From this aspect, it makes more sense to examine the algorithms not in a pure FLOPS-based context, but with regard to overall execution time. This way, speed of convergence is also taken into account, giving a more accurate measurement of the overall performance.

IV. EXPERIMENTAL EVALUATION

Our tests ran on a Playstation 3 gaming console, powered by a 3.2 GHz, 6-SPE Cell Broadband Engine processor, 256 MB XDR memory, running Debian/GNU Linux – kernel version 2.6.24, loaded with Cell SDK 3.1. Both algorithms were set up using $a = 0.5, dt = 1.0, dx = 2 \times a \times dt + 1 = 2$. We gradually improved each algorithm, measuring its performance in each step in order to demonstrate how different optimizations increase execution speed. For each result shown in following diagrams we ran the corresponding test three times and took the average value. Each test ran until it converged to a solution. Both absolute performance and total execution time to converge are measured.

A. Performance and time measurements

GFLOPS – Number of SPEs: In Figure 5, our measurements are compared with the theoretical values explained above. We can see that performance improvement is linear to the number of SPEs used, which implies that communication between SPEs takes up only a small percentage of total execution time. There is a deviation from the linear behavior when 5 SPEs are used in the in-place method; actually, this is expected, as each SPE is not assigned the same number of block-columns in a 6144×6144 grid, when we use 5 SPEs. This load imbalance is a disadvantage of the in-place method stemming from the elongated block shape.

Total steps required for convergence: Table I presents a comparison between in-place and out-of-place methods in terms of total iterations required for algorithms to converge to a solution of the partial differential equation. For all grid sizes, the in-place method needs almost half the iterations compared to the out-of-place method. Figure 6 depicts these results.

Total Execution Time – Number of SPEs: The diagram in Figure 7 shows how these two methods are compared to each other in terms of total time needed for convergence. Both methods, running in their fully optimized form, need almost the same time to converge, with the in-place method having a slight advantage. This observation is the result of the fact that the in-place method runs at almost half the speed of the out-of-place method, but also requires almost half the iterations in order to converge.

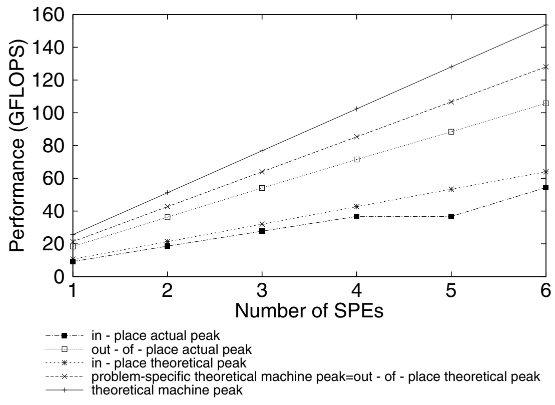

Figure 5. Performance model on a 6144×6144 grid

Grid Size	In-place	Out-of-place
512×512	1305	2232
1024×1024	2340	4410
2048×2048	4455	8595
3072×3072	6570	12735
4096×4096	8685	16875
6144×6144	12870	25155

Table I
TOTAL ITERATIONS REQUIRED FOR CONVERGENCE

B. In-place performance improvements

The in-place optimization distribution diagram (see Figure 8) shows how the in-place algorithm performance is gradually benefited by each successive optimization. The benefits of manually scheduling instructions are very clear to see. In the presence of all other 'conventional' optimizations, manual instruction scheduling can almost double

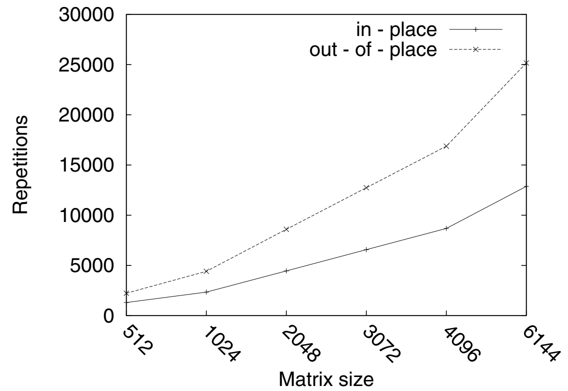

Figure 6. Comparison on total iterations required for convergence

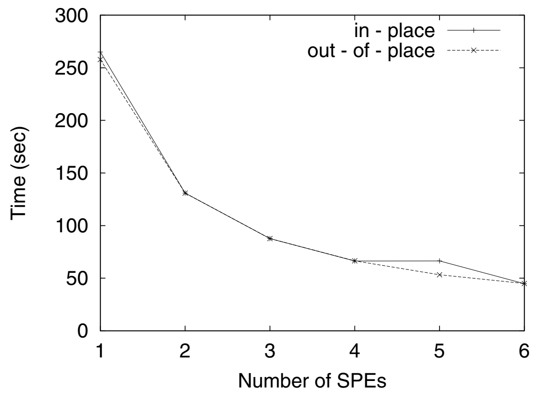

Figure 7. Total execution time required for convergence on a 6144×6144 grid

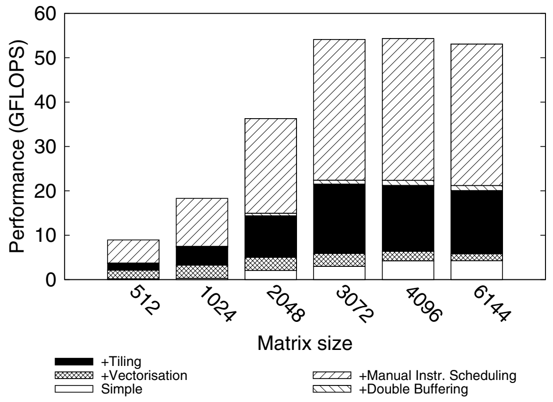

Figure 8. Performance scaling after each improvement in in-place algorithm using 6 SPEs

performance. Contribution of tiling is also worth mentioning.

C. Out-of-place performance improvements

The out-of-place optimization distribution diagram (see Figure 9) shows how the out-of-place algorithm performance is benefited as we gradually apply each optimization. Once again, manual instruction scheduling seems to be an essential part in achieving top GFLOPS measurements and application of tiling also has its merits.

V. Conclusions and future work

In this paper we worked on the problem of the efficient parallel execution of the two-dimensional advection equation using the Cell Broadband Engine, trying to exploit its particular architectural characteristics.

- Our experimental results have shown that the estimated overall time between these two algorithms is similar, with the in-place approach needing marginally fewer total computation time to converge. Regarding the ease of implementation, the out-of-place approach has the advantage that it can be implemented in a far simpler way thanks to the absence of limitations which we are faced with in the in-place algorithm. On the other hand, the code for the in-place approach has potential for further improvement. A more complex implementation, which simultaneously calculates two or even four time steps, could take full advantage of the powerful SPE pipelines, presumably reaching theoretical algorithm peak performance. In this sense, if programmability is excluded from the selection criteria, the in-place algorithm is the preferred choice due to better performance behavior.
- Based on the above findings, this work comes to support the general opinion that numerical criteria cannot be the sole guide for the selection of an efficient algorithm in a high-performance parallel architecture. Architectural characteristics and programmability play equally important roles and one needs to thoroughly evaluate all these factors to perform a successful selection.
- Block-major layout technique can substantially contribute to reduce communication overhead. A lot of concurrent DMA transfers, which is a common case when using DMA-lists, can easily congest EIB.
- The way we assigned blocks to SPEs (column-traversal), eliminating dependence vectors in the vertical direction along with the corresponding data transfers, seems to support well algorithm execution performance.
- Although multi-buffering techniques and data vectorization are the most 'advertised' features (apart from multicore design) of the CBE Architecture, it turns out that a great portion of CBE power lies in its heterogeneous pipelines and the ability to simultaneously execute heterogeneous types of instructions, which leads to executions where the floating-point arithmetic pipeline does not need to stall waiting for data to be loaded and formatted in registers.
- Modern compilers seem unable to produce high-

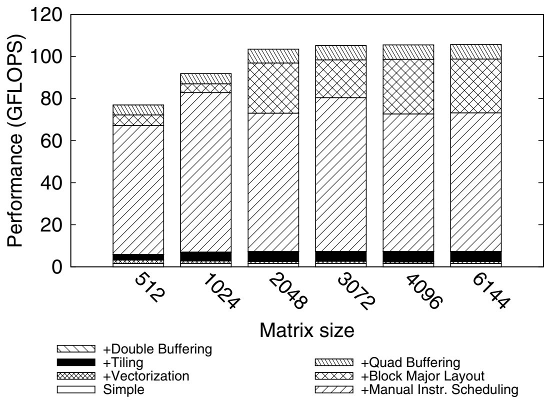

Figure 9. Performance scaling after each improvement in out-of-place algorithm using 6 SPEs

performance code that takes advantage of the heterogeneous pipelines. The developer either has to make complex and time-consuming manual code optimizations or compromise on poor performance. A better progress seems to have been made in compiler auto-vectorization, but there is still room for improvement.

There are various topics that remain open to further study. The most interesting among them are a double-precision implementation of the same problem, running on more advanced hardware (PowerXCell 8i), capable of exploiting similar potentials using double-precision arithmetic, implementation of solvers for the three-dimensional problem, study of other PDEs (e.g. the diffusion equation) and numerical schemes (e.g. multi-coloring schemes like Red-Black). Future work will also focus on techniques to achieve better instruction scheduling automatically.

Acknowledgments

This research is supported by the PENED 2003 Project (EPAN), co-funded by the European Social Fund (80%) and National Resources (20%).

References

[1] M. J. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of Computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[2] L. Peng, R. S. Seymour, K. ichi Nomura, R. K. Kalia, A. Nakano, P. Vashishta, A. Loddoch, M. Netzband, W. R. Volz, and C. C. Wong, "High-order stencil computations on multicore clusters," in *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009*, Rome, Italy, May 23–29 2009.

[3] G. I. Goumas, A. Sotiropoulos, and N. Koziris, "Minimizing completion time for loop tiling with computation and communication overlapping," in *Proceedings of the 15th International Parallel Distributed Processing Symposium (IPDPS-01)*, San Francisco, CA, April 23–27 2001.

[4] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, San Diego, California, United States, January 1988, pp. 319–329.

[5] P. Boulet, A. Darte, T. Risset, and Y. Robert, "(Pen-)Ultimate tiling," *Integration - The VLSI Journal*, vol. 17, no. 1, pp. 33–51, August 1994.

[6] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the third conference on Computing Frontiers*, Ischia, Italy, May 3–5 2006.

[7] S. I. S. IBM Systems & Technology Group, "Using cell broadband engine technology to improve molecular modeling applications," Route 100, Somers, New York 10589, U.S.A., Tech. Rep., 2008.

[8] *Cell Broadband Engine Programming Tutorial*, IBM, http://www-128.ibm.com/developerworks/power/cell/, 2007.

[9] *Cell Broadband Engine Programming Handbook*, 1st ed., IBM, http://www-128.ibm.com/developerworks/power/cell/, 2007.

[10] T. Chen, R. Raghavan, J. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation," IBM, http://www.ibm.com/developerworks/power/library/pa-cellperf/, Tech. Rep., November 2005.

[11] G. E. Karniadakis and R. M. Kirby, *Parallel Scientific Computing in C++ and MPI: A Seamless Approach to Parallel Algorithms and their Implementation.* Cambridge University Press, 2002.

[12] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and explicit optimizations for stencil computations," in *Proceedings of the 2006 workshop on Memory system performance and correctness*, San Jose, California, October 22 2006.

[13] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Conference of High Performance Networking and Computing, Proceedings of the 2008 ACM/IEEE conference of Supercomputing*, Austin, Texas, November 15–21 2008.

[14] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM Review (SIREV)*, vol. 51, no. 1, pp. 129–159, 2009.